Maintaining the Size of LZ77 on Semi-dynamic Strings

Hideo Bannai
M&D Data Science Center, Tokyo Medical and Dental University (TMDU), Japan

Panagiotis Charalampopoulos
Birkbeck, University of London, UK

Jakub Radoszewski
Institute of Informatics, University of Warsaw, Poland

Abstract

We consider the problem of maintaining the size of the LZ77 factorization of a string S of length at most n under the following operations: (a) appending a given letter to S and (b) deleting the first letter of S. Our main result is an algorithm for this problem with amortized update time $\tilde{\mathcal{O}}(\sqrt{n})$. As a corollary, we obtain an $\tilde{\mathcal{O}}(n\sqrt{n})$-time algorithm for computing the most LZ77-compressible rotation of a length-n string—a naive approach for this problem would compute the LZ77 factorization of each possible rotation and would thus take quadratic time in the worst case. We also show an $\Omega(\sqrt{n})$ lower bound for the additive sensitivity of LZ77 with respect to the rotation operation. Our algorithm employs dynamic trees to maintain the longest-previous-factor array information and depends on periodicity-based arguments that bound the number of the required updates and enable their efficient computation.

2012 ACM Subject Classification Theory of computation → Pattern matching; Theory of computation → Data compression

Keywords and phrases Lempel-Ziv, compression, LZ77, semi-dynamic algorithm, cyclic rotation

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.19

Funding *Hideo Bannai*: JSPS KAKENHI Grant Number JP20H04141.
Jakub Radoszewski: Supported by the Polish National Science Center, grant no. 2022/46/E/ST6/00463.

1 Introduction

Lempel-Ziv 77 (LZ77) [70] is one of the most well known and most effective compression algorithms that admit efficient implementations. An LZ-like parsing of a string is a partitioning of the string into *phrases*, where each phrase starting at position i is either a single letter that does not occur previously, or is a prefix of the rest of the string of length $\ell \geq 1$ that has a previous occurrence at some position $s < i$. Each phrase can then be encoded by a pair $(1, T[i])$, or (ℓ, s), depending on the type of the phrase. The latter is a reference to a previous occurrence of the phrase, and thus compression can be achieved when there are many long phrases. The LZ-like parsing is also known as a Lempel-Ziv-Storer-Szymanski factorization (LZSS) [63] with self-references or a C-factorization [14]. The number of phrases in an LZ-like parsing can be minimized by adopting a greedy left-to-right approach, which is the so-called LZ77 parsing, and can be computed off-line in $\mathcal{O}(n)$ time and space for linearly sortable alphabets, or in $\mathcal{O}(n \log \sigma)$ time and $\mathcal{O}(n)$ space for general ordered alphabets, where n is the length of the string and σ is the number of distinct letters in the string (also $o(n)$-time algorithms for well-compressible strings over a small alphabet are known [20, 36]). Algorithms for computing the LZ77 parsing of a given (static) string have been studied extensively [1, 18, 16, 54, 4, 31, 32, 38, 33, 27, 28, 69, 24, 53, 25, 57, 44].

In an *on-line* setting, where the string can grow by appending symbols at the end, only the last phrase of an LZ77 parsing can change. The LZ77 parsing of the string can be maintained in amortized $\mathcal{O}(\log \sigma)$ time for each append operation, by a direct adaptation of Ukkonen's suffix tree construction algorithm [67]. There are also results that focus on achieving smaller space [55, 62, 69, 58, 6].

The *fully-dynamic* setting, where edits to the text at any position are allowed, is much more challenging than the on-line setting. The efficient computation of the LZ77 parsing essentially relies on index data structures such as suffix trees or arrays, which allow fast prefix searches. Recent advances have showed that dynamic indices with poly-logarithmic update and query times are possible [53, 37], and together with dynamic longest common extension (LCE) queries [51, 52, 53], this enables the computation of the LZ77 parsing of a dynamic string S in $\tilde{\mathcal{O}}(|\mathsf{LZ}(S)|)$ time, where $|\mathsf{LZ}(S)|$ is the number of phrases in the LZ77 parsing of the current string. Note that this number can be linear in the size of the string.

In this paper, we consider the problem of maintaining the *size* of the LZ77 parsing of a *semi-dynamic* string of length at most n, where the allowed update operations are (a) appending a letter and (b) shrinking the string by deleting the first letter. We present an algorithm that processes each update in strongly sublinear time.

Related Work

Cormode and Muthukrishnan [13] introduced the *substring compression* problem, where the goal is to preprocess a static string so that given any factor of the string, the phrases in its LZ77 parsing can be computed efficiently. Existing solutions for the substring compression problem [13, 35, 42, 43] basically compute the LZ77 parsing one phrase at a time, and thus require $\tilde{\mathcal{O}}(|\mathsf{LZ}(S)|)$ time to answer a query for a factor S. Our aim is to achieve better query time when the size of the LZ77 parsing can be large, by considering the more restricted semi-dynamic setting, where the queried factor moves in a sliding-window fashion over a string (note, however, that the whole string is not necessarily given beforehand).

The notion of *compression sensitivity* [2] measures the degree to which the sizes of compressed representations can change in response to updates. Akagi et al. [2] considered the compression sensitivity of LZ77 under single-letter updates. We extend this result by showing that a cyclic rotation of a length-n string by one letter can change the size of the LZ77 parsing only by $\mathcal{O}(\sqrt{n})$, and there are arbitrarily long strings for which the size of the LZ77 parsing changes by $\Theta(\sqrt{n})$.

The semi-dynamic/sliding window setting has been considered for maintaining the suffix tree [12, 21, 46], the Directed Acyclic Word Graph [29, 60], as well as for other stringology problems [3, 15, 48, 47, 49]. Early studies of the dynamic longest common subsequence and edit distance problems considered models where the allowed updates are a subset of prepend, append, and delete the first or last letter, see [66, 45, 39, 30]. Our results are *not* related to the problem considered in [10], where the "sliding window" considered there is a range in which previous occurrences of the phrases are limited to those starting inside the range.

Our Contributions

We consider the following problem with strings indexed from 0.

Semi-dynamic LZ Compression Size
Maintained object: A string S of length at most n along with $\|\mathsf{LZ}(S)\|$.
Update: Perform one of the two following operations:
delete: $S \to S[1\,..\,|S|-1]$; $\quad$ append(a) for $a \in \Sigma$: $S \to Sa$.

Our main result can be stated as follows.

▶ **Theorem 1.** *The* Semi-dynamic LZ Compression *problem admits a solution using* $\mathcal{O}(n)$ *space and* $\mathcal{O}(\sqrt{n}\log^2 n)$ *amortized update time.*

We also define the problem in the sliding window model.

Sliding Window LZ Compression Size
Input: A string S of length n and an integer $d \in [1\,..\,n]$.
Output: The size of $\mathsf{LZ}(S[i\,..\,i+d))$, for each $i = 0, \ldots, n-d$.

We define the rotation operation on a string S as $\mathsf{rot}(S) = S[1\,..\,|S|)S[0]$. The string obtained from S by r applications of the rotation operation is denoted $\mathsf{rot}^r(S)$, while $\mathsf{rot}^0(S) := S$.

Most LZ-Compressible Rotation
Input: A string S of length n.
Output: An integer $r \in [0\,..\,n)$ such that $\mathsf{LZ}(\mathsf{rot}^r(S))$ has the least number of phrases, that is, $\arg\min_{r\in[0..n)} |\mathsf{LZ}(\mathsf{rot}^r(S))|$.

This problem is a special case of Sliding Window LZ Compression Size. It suffices to iterate over all length-n factors of string S^2 using a sliding window.

In Fact 25 in Section 6, we show that some rotation of S can have $\Theta(\sqrt{|S|})$ fewer phrases than S, or $\frac{2}{3}z$ phrases, where $z = |\mathsf{LZ}(S)|$. This implies that storing the best rotation value and compressing the rotation can yield better compression.

As a baseline, the Sliding Window LZ Compression Size can be solved using Generalized Substring Compression queries [35] as follows.

▶ **Proposition 2.** *The* Sliding Window LZ Compression Size *problem can be solved in* $\mathcal{O}(n\sqrt{\log n} + Z\log\log n)$ *time and* $\mathcal{O}(n\log\log n)$ *space, where* $Z = \sum_{i=0}^{n-d} |LZ(S[i\,..\,i+d))|$ *is the total number of phrases in the LZ77 factorizations of all length-d windows of* S*.*

Proof. Let us recall that in the Generalized Substring Compression problem, we are to preprocess a string S so that, given any factor $S[\ell\,..\,r]$ of S, we can compute the phrases in its LZ77 parsing efficiently. Keller et al. [35] presented an efficient data structure for answering Generalized Substring Compression queries based on range successor queries. If the later range successor data structure of Gao et al. [26] is used, we obtain an $\mathcal{O}(n\log\log n)$-size data structure that can be constructed in $\mathcal{O}(n\sqrt{\log n})$ time and can answer a Generalized Substring Compression query for $S[\ell\,..\,r]$ in $\mathcal{O}(|\mathsf{LZ}(S[\ell\,..\,r])|\log\log n)$ time. ◀

Other trade-offs in the proposition are also possible; see [50, 26, 19]. For example, one can obtain linear space with $\mathcal{O}(n\sqrt{\log n} + Z\log^\epsilon n)$ time, for any $\epsilon > 0$ [50, 26]. Still, the time complexity of the algorithm of Proposition 2 is $\tilde{\mathcal{O}}(n+Z)$, which can be as bad as $\tilde{\mathcal{O}}(n^2)$ for poorly compressible texts. As a corollary of Theorem 1, we obtain the following result.

▶ **Corollary 3.** *The* Sliding Window LZ Compression Size *and* Most LZ-Compressible Rotation *problems can be solved in* $\mathcal{O}(n\sqrt{n}\log^2 n)$ *time using* $\mathcal{O}(n)$ *space.*

Technical Overview

At the heart of our solution, lies the maintenance of a dynamic tree that encodes the longest-previous-factor array information; the LZ77 factorization corresponds to a single path in this tree and this path's length can be efficiently retrieved by maintaining said tree using

link-cut trees [61]. A periodicity-based argument allows us to show that deleting the first letter of S results in $\mathcal{O}(\sqrt{n})$ updates to our dynamic tree. Appending a letter to S might unfortunately lead to $\Omega(n)$ updates. However, all updated edges of the tree have the same target and consecutive sources. Moreover, the structure of those updates allows us to handle them efficiently in batches. Namely, we show that there are $\mathcal{O}(\sqrt{n})$ consecutive intervals of positions $[a\,..\,b]$ such that, for all elements i of each interval, the (rightmost) position of the longest previous factor starting at position i changes from some position j to some position j' and, for some integers x and y, for all $i \in [a\,..\,b]$, $i - j' = x$ and $i - j = y$. All in all, given the endpoints of the intervals and the changes in the (rightmost) positions of the longest previous factors, we update the tree structure in $\tilde{\mathcal{O}}(\sqrt{n})$ time in total by storing all edges with the same target using a joinable balanced binary search tree [64]. In order to exploit the above structural insights, we show that it suffices to maintain a data structure that allows us to efficiently retrieve the value $\mathsf{LPF}_S[i]$ for the elements of an $\tilde{\mathcal{O}}(\sqrt{n})$-size subset of $[1\,..\,|S|)$.

We obtain this data structure by exploiting ideas that stem from internal string queries, such as the Interval LCP problem [35]. For a static string S, it suffices to combine a suffix tree and a 2D range successor data structure over an $n \times n$ grid, where, for each suffix $S[j\,..\,n)$, we have a point $(\mathsf{RANK}[j], j)$, where $\mathsf{RANK}[j]$ is the lexicographic rank of said suffix among the suffixes of S. For computing the positions of longest previous factors, we enhance this data structure with further range successor data structures. We then maintain such a data structure that efficiently answers the desired queries when the involved factors are contained in $S[0\,..\,|S| - \mathcal{O}(\sqrt{n})]$. We overcome the technical challenge posed by the need to handle the remaining queries by analysing the periodic structure implied by "hard" such queries and batching them so that we only spend an additive $\mathcal{O}(\sqrt{n})$ factor overhead in the time complexity.

Structure of the paper. The auxiliary data structure for LPF computation in a semi-dynamic setting is described in Section 3. An abstract structure of the LPF-tree is defined in Section 4, where the periodicity-based arguments are also given. Implementations of operations on the tree are provided in Section 5. Finally, Section 6 considers the additive sensitivity of the size of the LZ77 parsing under a single rotation operation.

2 Preliminaries

Let $S = S[0]S[1]\cdots S[n-1]$ be a *string* (or *text*) of length $n = |S|$ over an integer alphabet Σ. The elements of Σ are called *letters*. For two positions i and j of S, we denote by $S[i\,..\,j]$ a string called a *factor* of S that starts at position i and ends at position j (the factor is empty, denoted by ε, if $i > j$). A factor of S can be represented in $\mathcal{O}(1)$ space by specifying the indices i and j of an occurrence of it. We define $S[i\,..\,j+1) = S[i\,..\,j] = S(i-1\,..\,j]$. A string U is a proper prefix (resp. suffix) of S if there exists a non-empty string V such that $S = UV$ (resp. $S = VU$).

If a string B is both a proper prefix and a proper suffix of a length-n string S, then B is called a *border* of S. A positive integer p is called a *period* of S if $S[i] = S[i+p]$ for all $i \in [0\,..\,n-p)$. String S has a period p if and only if it has a border of length $n-p$. We refer to the smallest period of S as *the period* of S, and denote it by $\mathsf{per}(S)$. String S is called *periodic* if $\mathsf{per}(S) \le n/2$.

▶ **Lemma 4** (Periodicity Lemma [22], weak version). *If p and q are periods of a string S and satisfy $p + q \le |S|$, then $\gcd(p, q)$ is also a period of S.*

▶ **Lemma 5** ([11, 56, 42]). *The set of occurrences of a string X in a string Y can be expressed as a union of $\mathcal{O}(|Y|/|X|)$ arithmetic progressions such that the difference of each progression equals $\mathsf{per}(X)$. The intervals spanned by the progressions are disjoint.*

For a string S of length n, we define the following arrays indexed from 0 to $n-1$:

$$\mathsf{LPF}_S[i] = \max(\{\ell \geq 0 \,:\, S[j\,..\,j+\ell) = S[i\,..\,i+\ell),\, j < i\} \cup \{0\})$$
$$\mathsf{LPFpos}_S[i] = \max(\{j < i \,:\, S[j\,..\,j+\mathsf{LPF}_S[i]) = S[i\,..\,i+\mathsf{LPF}_S[i]) \neq \varepsilon\} \cup \{-1\}).$$

See Figure 3 in Page 11 for an example.

▶ **Theorem 6** (Corollary of [8]). *For a string S of length n, arrays LPF_S and LPFpos_S can be constructed in $\mathcal{O}(n\sqrt{\log n})$ time.*

Proof. Array LPF_S can be constructed in $\mathcal{O}(n)$ time [17], while array LPFpos_S can be constructed in $\mathcal{O}(n(1+\log\sigma/\sqrt{\log n})) = \mathcal{O}(n\sqrt{\log n})$ time [8]. ◀

Predecessor data structures. For a static set, a combination of x-fast tries [68] and deterministic dictionaries [59] yields the following efficient deterministic data structure.

▶ **Fact 7** ([23, Proposition 2]). *A sorted static set $Y \subseteq [1\,..\,U]$ can be preprocessed in $\mathcal{O}(|Y|)$ time and space so that predecessor queries can be performed in $\mathcal{O}(\log\log|U|)$ time.*

A dynamic predecessor data structure over m integer keys can be stored using an exponential search tree [5] in $\mathcal{O}(m)$ space, supporting insertions, deletions, and predecessor queries in $\mathcal{O}(\log^2\log m/\log\log\log m)$ worst-case time.

3 Answering LPF Queries in a Batch in a Semi-dynamic String

In the semi-dynamic model, we will extensively use a solution to the following problem to compute previous occurrences of factors of the maintained string S.

Semi-dynamic Batch LPF
Maintained object: A string S of length at most n.
Update: Perform one of the two following operations:
- delete: $S \to S[1\,..\,|S|-1]$;
- append(a) for $a \in \Sigma$: $S \to Sa$.

Query: Given a set $Y \subseteq [0\,..\,|S|)$, compute $\mathsf{LPF}_S[y]$ and $\mathsf{LPFpos}_S[y]$ for each $y \in Y$.

This section is devoted to an $\mathcal{O}(n)$-space solution for this problem with $\mathcal{O}(\sqrt{n\log n})$ update time and $\mathcal{O}(\sqrt{n\log n} + |Y|\log^{\epsilon} n)$ query time, for any $\epsilon > 0$. We start with a discussion of static algorithms for computing LPF and LPFpos.

A solution to the following static problem can be used to answer queries for LPF in a sliding window if the whole string is known in advance.

Interval LCP
Input: A string T of length n.
Query: Given a factor F of T and an interval $[i\,..\,j]$, find the longest prefix F' of F that occurs in T at some position in $[i\,..\,j]$ and a position $k \in [i\,..\,j]$ such that $T[k\,..\,k+|F'|) = F'$.

▶ **Theorem 8** (Keller et al. [35], Belazzougui et al. [8]). *The* Interval LCP *problem admits a solution with $\mathcal{O}(n)$ space, $\mathcal{O}(n\sqrt{\log n})$ construction time, and $\mathcal{O}(\log^{\epsilon} n)$ query time, for any $\epsilon > 0$.*

To answer queries for LPFpos in a sliding window we would use an auxiliary problem called Interval LCP Position in which, in addition to the output of an Interval LCP query, we compute the rightmost position within the interval $[i\,..\,j]$ where the longest prefix of F occurs. Using range successor queries, one can obtain a solution for the Interval LCP Position problem with the complexities of Theorem 8.

Let us formally define *range successor* queries. We are given a set $\mathcal{P}$ of n points in an $n \times n$ grid. Given a range $[x\,..\,x'] \times [y\,..\,\infty)$ (or $[x\,..\,x'] \times (-\infty\,..\,y]$), we are to report a point of $\mathcal{P}$ that is included in the range and has a minimal y-coordinate (maximal y-coordinate, respectively). Clearly, the x and y coordinates can be interchanged in this definition.

▶ **Lemma 9.** *The* Interval LCP Position *problem admits a solution with $\mathcal{O}(n)$ space, $\mathcal{O}(n\sqrt{\log n})$ construction time, and $\mathcal{O}(\log^{\epsilon} n)$ query time, for any $\epsilon > 0$.*

Proof. We compute in $\mathcal{O}(n)$ time the suffix array SA, the rank array RANK and the LCP array [34]. Let us recall the definitions of these arrays. The suffix array $\mathsf{SA}[0\,..\,n)$ is a permutation of $[0\,..\,n)$ such that:

$$T[\mathsf{SA}[0]\,..\,n) < T[\mathsf{SA}[1]\,..\,n) < \cdots < T[\mathsf{SA}[n-1]\,..\,n);$$

then $\mathsf{SA}[\mathsf{RANK}[i]] = i$ for all $i \in [0\,..\,n)$. The LCP array is defined as follows:

$$\mathsf{LCP}[i] = \max\{\ell \geq 0 : T[\mathsf{SA}[i]\,..\,\mathsf{SA}[i]+\ell) = T[\mathsf{SA}[i+1]\,..\,\mathsf{SA}[i+1]+\ell)\} \text{ for } i \in [0\,..\,n-1).$$

Next, we perform $\mathcal{O}(n\sqrt{\log n})$-time preprocessing (see [8]) to construct an $\mathcal{O}(n)$-sized data structure for $\mathcal{O}(\log^{\epsilon} n)$-time range successor queries (see [50]) on two sets of points on $n \times n$ grids: set $\mathcal{P}_1 = \{(i, \mathsf{LCP}[i]) : i \in [0\,..\,n-1)\}$ and set $\mathcal{P}_2 = \{(i, \mathsf{RANK}[i]) : i \in [0\,..\,n)\}$; the set $\mathcal{P}_2$ was also used in [35]. Finally, we perform the preprocessing of Theorem 8.

Upon an Interval LCP Position query for a factor F and interval $[i\,..\,j]$, we first use an Interval LCP query to compute the longest prefix F' of F that occurs at some position in $[i\,..\,j]$ and a position $k \in [i\,..\,j]$ such that $T[k\,..\,k+|F'|) = F'$. We would like to compute the maximum index $k' \in [k\,..\,j]$ such that $T[k'\,..\,k'+|F'|) = F'$.

First, we use range successor queries on the set of points $\mathcal{P}_1$ to locate the range $[\ell\,..\,r]$ in the suffix array that contains all the suffixes that have a longest common prefix with suffix $T[k\,..\,n]$ of length at least $|F'|$. Namely, to compute r, we find the smallest x-coordinate of a point from $\mathcal{P}_1$ in the range $[\mathsf{RANK}[k]\,..\,\infty) \times [0\,..\,|F'|)$. If there is no such point, $r = n-1$, and otherwise r is the computed x-coordinate. The computation of ℓ is symmetric.

Now, among the suffixes that correspond to $\mathsf{SA}[\ell\,..\,r]$, we would like to find the suffix occurring at a maximum position that is at most j. We ask a range successor query on the set of points $\mathcal{P}_2$ to find the maximum y-coordinate of a point in $[\ell\,..\,r] \times (-\infty\,..\,j]$; the returned y-coordinate is the sought position k'.

The Interval LCP query and each range successor query take $\mathcal{O}(\log^{\epsilon} n)$ time, for any $\epsilon > 0$ [8, 35]. ◀

The data structure of Lemma 9 essentially consists of the suffix tree of T (which is used in the data structure underlying Theorem 8) and range successor data structures. The corollary below follows from the work of Keller et al. [35] and Lemma 9.

▶ **Corollary 10.** *A string T of length n can be preprocessed in $\mathcal{O}(n\sqrt{\log n})$ time so that, given a string F and an interval $[i\,..\,j]$, computing the longest prefix of F that occurs in T at some position in $[i\,..\,j]$, as well as the rightmost position in $[i\,..\,j]$ at which it occurs, reduces in $\mathcal{O}(\log^{\epsilon} n)$ time, for any $\epsilon > 0$, to computing the locus of L in the suffix tree of T, where L is the longest prefix of F that occurs in T.*

We are now ready to proceed to semi-dynamic computation of LPF and LPFpos.

▶ **Lemma 11.** *The Semi-dynamic Batch LPF problem admits an $\mathcal{O}(n)$-space solution with $\mathcal{O}(\sqrt{n\log n})$ update time and $\mathcal{O}(\sqrt{n\log n} + |Y|\log^{\epsilon} n)$ query time, for any $\epsilon > 0$.*

Proof. We will be rebuilding some data structures over the string S after every $\lfloor\sqrt{n}\rfloor$ updates. We will keep the update-time bound worst-case by using the so-called time slicing technique, that is, splitting the work required for the construction of the data structure into roughly equal chunks and distributing them among the subsequent $\lfloor\sqrt{n}\rfloor$ updates; if the data structures can be constructed in $\tilde{\mathcal{O}}(n)$ time, then we will spend $\tilde{\mathcal{O}}(\sqrt{n})$ time on each single update. Let S_1 be the current string S, S_2 be S_1 after $\lfloor\sqrt{n}\rfloor$ updates, and S_3 be S_2 after $\lfloor\sqrt{n}\rfloor$ updates. The data structure for S_1 will be ready by the time S_2 is processed and will be used until S_3 is reached, at which point the data structure that is constructed for S_2 will be ready. This way, when processing the current string S, we can assume that we have access to data structures for a string $Z = VS[0\,..\,|S| - x)$, where $x \le 2\sqrt{n}$ and V is a string composed of all deleted letters since the construction of this data structure was issued.

The data structures that are stored for Z include the static data structure of Corollary 10 for the Interval LCP Position problem that takes $\mathcal{O}(n)$ space and $\mathcal{O}(n\sqrt{\log n})$ time to construct; the suffix tree of Z augmented in $\mathcal{O}(n)$ time with the weighted-ancestor-queries data structure of [7], which allows one to retrieve in $\mathcal{O}(1)$ time the locus of any given factor of Z in the suffix tree of Z, and other $\mathcal{O}(n)$-time constructible data structures based on the suffix tree of Z to be specified later.

Let us now discuss how to compute $\mathsf{LPF}_S[y]$ and $\mathsf{LPFpos}_S[y]$ for all $y \in Y$. We compute at most three candidate values for $\mathsf{LPF}_S[y]$ for each $y \in Y$, together with candidate positions $\mathsf{LPFpos}_S[y]$, and we take the maximum LPF value and the corresponding position in the end.

Case I: $\mathsf{LPFpos}_S[y] \ge d := |S| - 10\lceil\sqrt{n}\rceil$, so $y > d$. We compute the $\mathsf{LPF}_{S'}$ and $\mathsf{LPFpos}_{S'}$ arrays for $S' = S[d\,..\,|S|)$ in $\mathcal{O}(\sqrt{n\log n})$ time (Theorem 6). For each $y \in Y$ with $y > d$, we have a candidate $\mathsf{LPF}_{S'}[y-d]$ for $\mathsf{LPF}_S[y]$ along with candidate $\mathsf{LPFpos}_{S'}[y-d]$ for $\mathsf{LPFpos}_S[y]$.

Case II: $\mathsf{LPFpos}_S[y] + \mathsf{LPF}_S[y] < |S| - x$. (Let us note that, however, $y + \mathsf{LPF}_S[y]$ can be as large as $|S|$ in this case.) Our main aim is to compute, for each $y \in Y$, the locus of the longest prefix L_y of $S[y\,..\,|S|)$ that occurs in $S[0\,..\,|S| - x)$. Let the elements of Y in increasing order be $y_0, y_1, \ldots$; we process them in this order. Starting from the locus of $S[y_0\,..\,|S| - x)$ which we compute in $\mathcal{O}(1)$ time using a weighted ancestor query, we go down in the suffix tree letter by letter with the aim of computing L_{y_0}. Note that we follow an edge as long as the node we reach corresponds to a factor of $S[0\,..\,|S| - x)$ (and not just of Z); this can be checked in constant time after a linear-time bottom-up preprocessing of the suffix tree. When we have found the locus of L_{y_0} we do the following. From the suffix tree, we obtain an index i such that $L_{y_0} = Z[i\,..\,i + |L_{y_0}|)$. In constant time, using a weighted ancestor query, we go to the locus of $Z[i + (y_1 - y_0)\,..\,i + |L_{y_0}|)$ and start the search for L_{y_1} from there; and so on. We process $|Y|$ suffixes and only have $x = \mathcal{O}(\sqrt{n})$ letters to extend them by. The total time required for the described process is thus $\mathcal{O}(|Y| + \sqrt{n}\log\log n)$ assuming that the children

of a node in the suffix tree of Z are stored using the predecessor data structure of Fact 7. We then use Corollary 10 to compute a pair of candidates for $\mathsf{LPF}_S[y]$ and $\mathsf{LPFpos}_S[y]$ for each $y \in Y$, spending $\mathcal{O}(\log^\epsilon n)$ time for each y.

Case III: $\mathsf{LPFpos}_S[y] < |S| - 10\sqrt{n}$ and $\mathsf{LPFpos}_S[y] + \mathsf{LPF}_S[y] \geq |S| - x$. We have

$$y \leq |S| - \mathsf{LPF}_S[y] \leq \mathsf{LPFpos}_S[y] + x \leq \mathsf{LPFpos}_S[y] + 2\sqrt{n}, \text{ and}$$

$$\mathsf{LPF}_S[y] \geq |S| - x - \mathsf{LPFpos}_S[y] > 10\sqrt{n} - x \geq 8\sqrt{n}.$$

Hence, factors $S[y\,..\,y + \mathsf{LPF}_S[y])$ and $S[\mathsf{LPFpos}_S[y]\,..\,\mathsf{LPFpos}_S[y] + \mathsf{LPF}_S[y])$ start at most $2\sqrt{n}$ positions apart and overlap by more than $6\sqrt{n}$ positions. This means that $S[\mathsf{LPFpos}_S[y]\,..\,y + \mathsf{LPF}_S[y])$ is periodic with period at most $2\sqrt{n}$. In particular, due to the periodicity lemma (Lemma 4), the period of this factor must be equal to the period of $S[|S| - \lfloor 8\sqrt{n}\rfloor\,..\,|S| - x)$.

We can compute the period p of $S[|S| - \lfloor 8\sqrt{n}\rfloor\,..\,|S| - x)$ in $\mathcal{O}(\sqrt{n})$ time using the Morris-Pratt algorithm [41]. If $p \leq 2\sqrt{n}$, we compute the maximal factor $S[\ell\,..\,r]$ of S that contains $S[|S| - \lfloor 8\sqrt{n}\rfloor\,..\,|S| - x)$ and has the same period; the periodicity can be extended to the left in constant time after a linear-time preprocessing of Z (using longest common extension queries [9]) and to the right in $\mathcal{O}(\sqrt{n})$ time using letter comparisons. Then, for each $y \in [\ell + p\,..\,r]$, we have a candidate $r - y + 1$ for $\mathsf{LPF}_S[y]$ along with candidate $y - p$ for $\mathsf{LPFpos}_S[y]$. ◀

▶ Remark 12. For the purposes of Sliding Window LZ Compression Size problem for a string S, instead of Lemma 11, one could simply build the data structure encapsulated in Lemma 9 for S and use it to compute all required values $\mathsf{LPF}_S[y]$ and $\mathsf{LPFpos}_S[y]$ in the implied instance of the Semi-dynamic Batch LPF problem.

4 Properties of Longest Previous Factors and LPF-Tree

We next show two properties of the LPF and LPFpos arrays. The first of them bounds the number of zeroes in the LPFpos_U array.

To prove the lemma, we show that the values $\mathsf{LPF}_U[i]$ for increasing positions i such that $\mathsf{LPFpos}_U[i] = 0$ are strictly increasing. This, together with the fact that no three factors of the form $U[i\,..\,i + \mathsf{LPF}_U[i])$ for these positions overlap, shows that there are $\mathcal{O}(\sqrt{n})$ such positions. The aforementioned fact follows by periodicity.

▶ **Lemma 13.** *For a string U of length n, the number of positions $i \in [0\,..\,n)$ such that $\mathsf{LPFpos}_U[i] = 0$ is $\mathcal{O}(\sqrt{n})$.*

Proof. Let $I = \{i \in [1\,..\,n) : \mathsf{LPFpos}_U[i] = 0\}$. First, let us note that, for any $i, i' \in I$ with $i < i'$, we have $\mathsf{LPF}_U[i] < \mathsf{LPF}_U[i']$. Indeed, if we had $\mathsf{LPF}_U[i] \geq \mathsf{LPF}_U[i']$, then, for $\ell = \mathsf{LPF}_U[i']$, we would have $U[i\,..\,i + \ell) = U[0\,..\,\ell) = U[i'\,..\,i' + \ell)$, which would imply $\mathsf{LPFpos}_U[i'] \geq i > 0$, yielding a contradiction.

By the above, to prove the statement of the lemma, it suffices to show that if $i \in I$, then there is at most one element $i' \in I \cap (i\,..\,i + \lfloor \frac{1}{2}\mathsf{LPF}_U[i]\rfloor]$.

Assume that such elements i and i' exist. The setting is illustrated in Figure 1. We will show that then i' is determined uniquely for i. For $\ell = \mathsf{LPF}_U[i]$, we have that $U[i\,..\,i + \ell) = U[i'\,..\,i' + \ell)$ is a border of $U[i\,..\,i' + \ell)$. Hence, $U[i\,..\,i' + \ell)$ has a period $p := i' - i \leq \ell/2$ and is thus periodic. Let q be the smallest period of $U[i\,..\,i' + \ell)$. By the periodicity lemma (Lemma 4), q divides p.

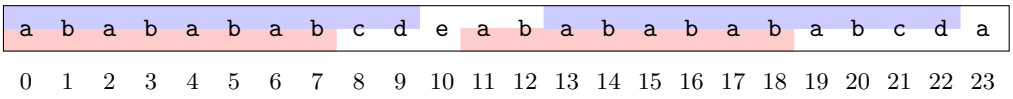

Figure 1 An illustration of the proof of Lemma 13. For $i = 11$ and $i' = 13$, we have $\mathsf{LPFpos}_U[i] = \mathsf{LPFpos}_U[i'] = 0$, $\ell = \mathsf{LPF}_U[i] = 8$ and $\ell' = \mathsf{LPF}_U[i'] = 10$. In this example, we have $q = 2$.

By definition, $U[i\,..\,i+\ell) = U[0\,..\,\ell)$, so $U[0\,..\,\ell)$ has period q. Moreover, $U[i+\ell] = U[i+\ell-q] = U[\ell-q]$, where the first equality follows from the fact that position $i+\ell$ is within the factor $U[i\,..\,i'+\ell)$. Therefore, $U[\ell] \neq U[\ell-q]$, because otherwise we would have $U[i\,..\,i+\ell] = U[0\,..\,\ell]$ and $\mathsf{LPF}_U[i] > \ell$. This means that $U[0\,..\,\ell]$ does *not* have period q, i.e., $U[i'\,..\,i'+\ell]$ does not have period q (as $\mathsf{LPF}_U[i'] > \ell$).

Let r be the smallest position such that $r \geq i+q$ and $U[i\,..\,r]$ does not have period q. We must have $i' = r - \ell$. All in all, i' is uniquely determined by i in U. ◀

▶ Remark 14. The bound from Lemma 13 is tight. Let $a_1, \ldots, a_m$ be distinct letters and consider strings $S_i = a_1 a_2 \cdots a_i$. Then the string $U = S_m S_1 S_2 \cdots S_m$ has length $\Theta(m^2)$ and for each starting position $j > 0$ of some S_i, for $i = 1, \ldots, m$, we have $\mathsf{LPFpos}_U[j] = 0$.

Let us define $\mathsf{LPFpos}'_S[i] = i - \mathsf{LPFpos}_S[i]$.

The next lemma characterizes the positions i such that $i + \mathsf{LPF}_U[i] = |U|$. There can be many such positions, even $\Theta(n)$, say, for a unary string $U = \mathtt{a}^n$. However, there are only $\mathcal{O}(\sqrt{n})$ different values $\mathsf{LPFpos}'_U[i]$ for such positions. Here, we need to consider the LPFpos'_U array and not the LPFpos_U array, as the latter can have $\Theta(n)$ different values for the positions of the considered type; the unary string $U = \mathtt{a}^n$, for which $\mathsf{LPFpos}_U[i] = i - 1$ for each $i \in [0\,..\,n)$, is an example.

In the proof it suffices to consider positions $i \leq n - \sqrt{n}$ such that $i + \mathsf{LPF}_U[i] = |U|$. Each such position implies an occurrence of a length-$\lfloor\sqrt{n}\rfloor$ suffix V of U in V. In turn, the occurrence of V determines the value of $\mathsf{LPFpos}'_U[i]$. We consider all possible occurrences of V in U as $\mathcal{O}(\sqrt{n})$ arithmetic progressions (cf. Lemma 5) and use periodicity to show that $\mathcal{O}(1)$ occurrences in each progression can be implied in the aforementioned sense.

▶ **Lemma 15.** *For a string U of length n, among all positions $i \in [0\,..\,n)$ for which $i + \mathsf{LPF}_U[i] = n$, there are $\mathcal{O}(\sqrt{n})$ different values $\mathsf{LPFpos}'_U[i]$.*

Proof. We denote $s = n - \lfloor\sqrt{n}\rfloor$. Obviously, there are at most $\lfloor\sqrt{n}\rfloor - 1$ different values $\mathsf{LPFpos}'_U[i]$ for $i \in (s\,..\,n)$.

Let $V = U[s\,..\,n)$. Note that if $\mathsf{LPFpos}_U[i] = j$ for some $i \in [0\,..\,s]$ with $i + \mathsf{LPF}_U[i] = n$, then there is an occurrence of V in U at position $j + (s - i)$. In this case, we say that the occurrence of V in U at position $j + (s - i)$ is *implied* by position i or that position i *implies* the occurrence.

By Lemma 5, the set of occurrences of V in U consists of $\mathcal{O}(\sqrt{n})$ maximal arithmetic progressions with common difference $\mathsf{per}(V)$. Let J be one of these arithmetic progressions. We will show that there are at most two elements $p \in J$ such that the occurrence of V at position p is implied by any position $i \in [0\,..\,s]$. This will conclude the proof, as for all positions i that imply an occurrence of V at position p, the value $\mathsf{LPFpos}'_U[i] = s - p$ is the same. Let $U[n-d\,..\,n)$ be the longest suffix of U with period $\mathsf{per}(V)$. Figure 2 contains an illustration of possible cases considered below.

Let J_0 be the arithmetic progression containing position s. Assume first that position $i \in [0\,..\,s]$ implies an occurrence of V at a position p in $J = J_0$. We must have $|J_0| > 1$. If $i \geq n - d$, then $U[i\,..\,n)$ has period $\mathsf{per}(V)$. We have $U[j\,..\,j + \mathsf{LPF}_U[i]) = U[i\,..\,n)$, so

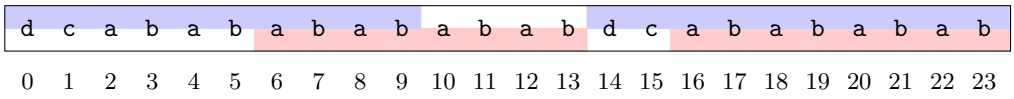

Figure 2 An illustration of the proof of Lemma 15. Let $V = \mathtt{abab}$. We have $n - d = 16$ and $\{i \in [0\,..\,23] : i + \mathsf{LPF}_U[i] = 24\} = [14\,..\,23]$. Let $J = \{2, 4, 6, 8, 10\}$ be an arithmetic progression of occurrences of V in U. Positions 14 and 15 imply the occurrence of V at position 6; the corresponding equality $U[0\,..\,9] = U[14\,..\,23]$ is illustrated using the blue top rectangles. Positions 16 and 17 imply the occurrence of V at position 10; the corresponding equality $U[6\,..\,13] = U[16\,..\,23]$ is illustrated using the red bottom rectangles. For each position $i \in [18\,..\,23]$, we have $\mathsf{LPFpos}'_U[i] = \mathsf{per}(V) = 2$.

$j \geq n - d$, as otherwise $U[j\,..\,j + \mathsf{LPF}_U[i])$ would not have period $\mathsf{per}(V)$. One cannot have $j \in (i - \mathsf{per}(V)\,..\,i)$, as this would imply an additional occurrence of V that is not in the progression. We always select the rightmost position, so $p = s - \mathsf{per}(V)$ is determined uniquely (and $j = i - \mathsf{per}(V)$). Now we need to note that the case that $i < n - d$ is impossible. Indeed, in this case the longest suffix of $U[i\,..\,n)$ that has period $\mathsf{per}(V)$ has length d, the longest suffix of $U[j\,..\,j + \mathsf{LPF}_U[i])$ that has period $\mathsf{per}(V)$ has length $d - (i - j)$, i.e., smaller than d, but $U[j\,..\,j + \mathsf{LPF}_U[i]) = U[i\,..\,n)$.

Henceforth we assume that $J \neq J_0$. For $r = \max J + |V|$, let $U[r - d'\,..\,r)$ be the longest suffix of $U[0\,..\,r)$ with period $\mathsf{per}(V)$. Assume that position $i \in [0\,..\,s]$ implies an occurrence of V at a position in J. If $n - i \leq \min(d, d')$, we have $i \in [n - d\,..\,n)$ and $U[i\,..\,n) = U[\max J + |V| - (n - i)\,..\,\max J + |V|)$. Hence, since $\mathsf{LPFpos}_U[i]$ stores the rightmost value in case of ties, the implied occurrence of V is the one starting at position $\max J$. Otherwise, the factor equality implied by $i + \mathsf{LPF}_U[i] = n$ means that $d \leq d'$. Thus we have $i < n - d$ and $U[i\,..\,n)$ does not have period $\mathsf{per}(V)$. Now, there is exactly one position $p \in J$ such that $U[p + |V| - d\,..\,p + |V|)$ has period $\mathsf{per}(V)$, but $U[p + |V| - d - 1\,..\,p + |V|)$ does not have this period. Namely, $p = \max J - (d' - d)$ and we must have $d' \equiv d \pmod{\mathsf{per}(V)}$. The occurrence of V at position p is the one implied by position i in this case. This concludes the proof that at most two occurrences of V in J can be implied by any position $i \in [0\,..\,s]$, and hence the whole proof. ◀

▶ Remark 16. The bound from Lemma 15 is tight. Let $a_1, \ldots, a_m$ be distinct letters and consider strings $S'_i = a_i a_{i-1} \cdots a_1$. Then the string $U = S'_m S'_{m-1} \cdots S'_1 S'_m$ has length $\Theta(m^2)$, $j + \mathsf{LPF}_U[j] = |U|$ for all $j \in [|U| - m\,..\,|U|)$ and all values $\mathsf{LPFpos}'_U[j]$, for $j \in [|U| - m\,..\,|U|)$, are different.

4.1 LPF-tree

Let us define $\mathsf{LPF}'_S[i] = \max(1, \mathsf{LPF}_S[i])$. We define an *LPF-tree* for a string S as a tree with nodes $[0\,..\,|S|]$, among which $|S|$ is the root, and edges $\{(i, i + \mathsf{LPF}'_S[i]) : i = 0, \ldots, |S| - 1\}$. The single edge outgoing from i has *label* equal to $\mathsf{LPFpos}'_S[i]$; see Figure 3.

From Theorem 6 we obtain the following.

▶ **Corollary 17.** *The LPF-tree of a string S of length n can be constructed in $\mathcal{O}(n\sqrt{\log n})$ time.*

The LZ77 parsing of a string S can be computed straightforwardly from the LPF_S and LPFpos_S arrays in a greedy manner; cf. [16, 17]. We make the following simple observation.

▶ **Observation 18.** *Let π be the 0-to-$|S|$ path in the LPF-tree of S. Then, $|LZ(S)|$ equals the number $|\pi|$ of edges of π. Additionally, for $k \leq |\pi|$, if the k-th edge on π is $(i, i + LPF'_S[i])$, then the k-th phrase of $LZ(S)$ is equal to $S[i\,..\,i + LPF'_S[i])$.*

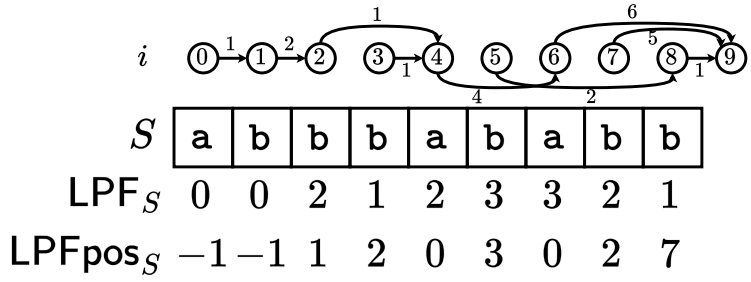

Figure 3 Example of an LPF-tree for the string `abbbababb`.

We need to extend the definition of an LPF-tree to the semi-dynamic setting. Let U be a semi-dynamic string that was obtained from an initial string by using $a = \mathsf{deletions}(U)$ first-letter deletions and some number of append operations. The nodes and edges of a *semi-dynamic LPF-tree* for U are, respectively, $[a\,..\,a+|U|]$, among which $a+|U|$ is the root, and $\{(i, i+\mathsf{LPF}'_U[i-a]) : i \in [a\,..\,a+|U|)\}$. The single edge outgoing from i has label equal to $\mathsf{LPFpos}'_U[i-a]$.

▶ **Lemma 19.** *Let us consider any string U and the semi-dynamic LPF-tree T of U. The sources of all edges of T that enter a given node form a set of consecutive integers.*

Proof. Let $a = \mathsf{deletions}(U)$. It suffices to show that for any $i \in [0\,..\,|U|-2]$, we have $i+\mathsf{LPF}'_U[i] \le i+1+\mathsf{LPF}'_U[i+1]$ (equivalently, $i+a+\mathsf{LPF}'_U[i] \le i+1+a+\mathsf{LPF}'_U[i+1]$).

The conclusion is obvious if $\mathsf{LPF}'_U[i]=1$. Hence, we assume that $\mathsf{LPF}'_U[i]>1$, so $\mathsf{LPF}'_U[i] = \mathsf{LPF}_U[i]$. Let $\ell = \mathsf{LPF}_U[i]$ and $p = \mathsf{LPFpos}_U[i] < i$. We have $U[p\,..\,p+\ell) = U[i\,..\,i+\ell)$ and hence $U[p+1\,..\,p+\ell) = U[i+1\,..\,i+\ell)$. Consequently, $i+1+\mathsf{LPF}'_U[i+1] \ge i+\ell$, as required. ◀

5 Updating a Semi-dynamic LPF-tree

We start by introducing an abstract data structure that will be used to store labels of edges of an LPF-tree. The data structure stores key-value pairs such that, for each pair, the value is smaller than the key. The set of keys at any time is a set of consecutive integers and is denoted by I. The size of I is denoted by n and the set of values is a subset of $[0\,..\,n)$. The following operations are supported:

(a) insertion of a pair with key $\max I+1$, where I is the current interval of keys, or deletion of a pair with key $\min I$,

(b) setting the value of all keys in a specified interval $[k_1\,..\,k_2]$ to a given integer v, knowing that their values were all equal to an integer v',

(c) reporting all key-value pairs for which the difference between the key and the value is minimal.

Let us call this data structure $\mathcal{D}$. Let $\pi_n = (\log\log n)^2/\log\log\log n$.

▶ **Lemma 20.** *Data structure $\mathcal{D}$ can be implemented in $\mathcal{O}(n)$ space so that each operation* (a), (b) *is performed in $\mathcal{O}(\pi_n)$ time and each pair in operation* (c) *is reported in $\mathcal{O}(\log\log n)$ time.*

Proof. For each value, we store the set of keys with this value as a collection of maximal intervals in a dynamic predecessor data structure. The predecessor data structures are stored in dynamic predecessor data structure indexed by values 0 through $n-1$.

Moreover, for each value, the minimum key with this value is determined and such key-value pairs are stored in a min-type priority queue ordered by the difference between the key and the value. Whenever an operation is performed on one of the predecessor data structures, the priority queue is updated accordingly.

Insertion in operation (a) requires to insert a singleton interval $\{\max I + 1\}$ to the predecessor data structure for its value, possibly merging the interval with the previous one. Deletion in operation (a) requires to remove the first element of the first interval in the predecessor data structure for this value.

In operation (b) we identify the (at most one) interval I in the predecessor data structure for value v' that contains $[k_1 \,..\, k_2]$ as a sub-interval. Then I is replaced by the at most two intervals $I \setminus [k_1 \,..\, k_2]$, and the interval $[k_1 \,..\, k_2]$ is inserted into the predecessor data structure for v, possibly being merged with any adjacent intervals.

In operation (c), elements are removed from the priority queue one by one and reported until the next element has a different priority. Afterwards they are reinserted to the priority queue.

We use the dynamic predecessor data structure [5] that requires $\mathcal{O}(m)$ space on m elements and supports queries in $\mathcal{O}(\pi_n)$ time. The priority queue requires $\mathcal{O}(\log\log n)$ time per operation [65]. ◀

Let U be a semi-dynamic string with $\mathsf{deletions}(U) = a$. The labels of edges of a semi-dynamic LPF-tree for U will be stored as a set of key-value pairs with keys $i \in [a \,..\, a + |U|)$ and values $\mathsf{LPFpos}'_U[i-a]$ using data structure $\mathcal{D}$. Only edges for which $\mathsf{LPFpos}_U[i-a] \neq -1$ are stored in $\mathcal{D}$.

In the two lemmas below, we use the data structure $\mathcal{D}$ together with the Semi-dynamic Batch LPF data structure (Lemma 11) to efficiently compute the updates that need to be performed on the semi-dynamic LPF-tree upon each of the single-letter updates on the string considered in the semi-dynamic setting. The actual operations on the LPF-tree will be performed in Section 5.1 when we define the data structure representing the LPF-tree.

▶ **Lemma 21.** *Let U be a semi-dynamic string of length n and U' be U after the deletion of its first letter. The semi-dynamic LPF-tree for U' can be obtained from the semi-dynamic LPF-tree for U by updating $\mathcal{O}(\sqrt{n})$ edges.*

The set of edges to be updated, as well as their new labels, can be computed in $\mathcal{O}(\sqrt{n\log n})$ time. Data structure $\mathcal{D}$ can be updated in $\mathcal{O}(\sqrt{n}\pi_n)$ time.

Proof. Let $a = \mathsf{deletions}(U)$. First, the edge from a needs to be removed. No operation on $\mathcal{D}$ is required, as $\mathsf{LPFpos}_U[0] = -1$.

Then, let us note that if $\mathsf{LPFpos}_U[i-a] > 0$ for $i \in (a \,..\, a+n)$, then $\mathsf{LPF}'_U[i-a] = \mathsf{LPF}'_{U'}[i-(a+1)]$ and $\mathsf{LPFpos}_U[i-a]+a = \mathsf{LPFpos}_{U'}[i-(a+1)]+(a+1)$, so $\mathsf{LPFpos}'_U[i-a] = \mathsf{LPFpos}'_{U'}[i-(a+1)]$. Hence, only edges $(i, i+\mathsf{LPF}'_U[i-a])$ with $\mathsf{LPFpos}_U[i-a] = 0$, i.e., $\mathsf{LPFpos}'_U[i-a] = i-a$, remain to be updated. The bound on the number of such edges follows by Lemma 13.

The edges to be updated can be retrieved in $\mathcal{O}(\sqrt{n}\log\log n)$ time using data structure $\mathcal{D}$. Indeed, for such an edge, the difference of the key and the value of the corresponding pair in $\mathcal{D}$ satisfies $i - \mathsf{LPFpos}'_U[i-a] = a$. Moreover, if $\mathsf{LPFpos}_U[i-a] > 0$ for $i \in (a \,..\, a+n)$, then $i - \mathsf{LPFpos}'_U[i-a] = a + \mathsf{LPFpos}_U[i-a] > a$. Therefore, the sought edges can be obtained via operation (c) on data structure $\mathcal{D}$.

For each edge $(i, i+\mathsf{LPF}'_U[i-a])$ with label $\mathsf{LPFpos}'_U[i-a] = i-a$ that we remove, we have to insert edge $(i, i+\mathsf{LPF}'_{U'}[i-(a+1)])$ with label $\mathsf{LPFpos}'_{U'}[i-(a+1)]$. (The edge is inserted to $\mathcal{D}$ as well only if $\mathsf{LPFpos}_{U'}[i-(a+1)] \neq -1$.) We compute the targets and

labels of these edges in $\mathcal{O}(\sqrt{n \log n})$ time using Lemma 11. According to Lemma 20, data structure $\mathcal{D}$ can be updated in $\mathcal{O}(\sqrt{n}\pi_n)$ total time via $\mathcal{O}(\sqrt{n})$ calls to operation (b), each with a singleton interval. ◀

▶ **Lemma 22.** *Let U be a semi-dynamic string of length n and $U' = Uc$, for some letter c. The semi-dynamic LPF-tree for U' can be obtained from the semi-dynamic LPF-tree for U by adding a new root and an edge from the old root to the new root, as well as redirecting some number of edges with consecutive sources that lead to the old root to point to the new root.*

The set of edges to be updated, represented as $\mathcal{O}(\sqrt{n})$ groups of edges with consecutive sources, equal old label and equal new label, can be computed in $\mathcal{O}(\sqrt{n}\log^{1.5} n)$ time. Data structure $\mathcal{D}$ can be updated in $\mathcal{O}(\sqrt{n}\pi_n)$ time.

Proof. Let $a = \mathsf{deletions}(U)$. We first create a new node $a+n+1$, designate it to be the root, and insert an edge $(a+n, a+n+1)$ with label decided by an invocation of Lemma 11 in $\mathcal{O}(\sqrt{n \log n})$ time. (The edge is inserted to $\mathcal{D}$ if only $\mathsf{LPFpos}_{U'}[n] \neq -1$.) Then, since

$$i + \mathsf{LPF}_U[i-a] \leq i + \mathsf{LPF}_{U'}[i-a] \leq i + \mathsf{LPF}_U[i-a] + 1,$$

for all i, all we need to do is compute the nodes i such that $i + \mathsf{LPF}_U[i-a] = a+n$ and $i + \mathsf{LPF}_{U'}[i-a] = a+n+1$. Due to Lemma 19, these nodes form a set R of consecutive integers and, in particular, $R = (r\,..\,a+n)$ for some $r \in [a\,..\,a+n)$.

We can compute r in $\mathcal{O}(\sqrt{n}\log^{1.5} n)$ time using binary search and Lemma 11. It remains to partition R into sub-intervals with the same $\mathsf{LPFpos}'_{U'}$ and LPFpos'_U values. Note that the $\mathsf{LPFpos}'_{U'}$ and LPFpos'_U values in R are non-decreasing since an occurrence of a length-ℓ suffix of U or U' at a position p, implies an occurrence of any suffix of length $\ell - \mu$ for a positive integer μ at position $p + \mu$.

By Lemma 15, there are $\mathcal{O}(\sqrt{n})$ possible values of $\mathsf{LPFpos}'_{U'}[i]$, for $i \in R$. We next show how to compute the partition of R by values $\mathsf{LPFpos}'_{U'}[i]$ in $\tilde{\mathcal{O}}(\sqrt{n})$ time using Lemma 11. We maintain a set of disjoint active intervals whose union is the set of positions for which we have not yet computed the value of $\mathsf{LPFpos}'_{U'}$. Initially, our set of active intervals is $\{R\}$. Then, until there are no active intervals left, for each active interval J, in the order of decreasing size, we do the following. We compute $\mathsf{LPFpos}'_{U'}$ for $\min J$, $\max J$, and the midpoint $j = \lfloor (\min J + \max J)/2 \rfloor$ of J. If the interval is of size at most three, we partition it to three singletons which are marked as inactive and labeled with the corresponding $\mathsf{LPFpos}'_{U'}$ values. Else, for the most distant $x, y \in \{\min J, j, \max J\}$ such that $\mathsf{LPFpos}'_{U'}[x-a] = \mathsf{LPFpos}'_{U'}[y-a]$, if they exist, we designate $[x\,..\,y]$ as inactive and label it with $\mathsf{LPFpos}'_{U'}[x-a]$. The remaining positions yield at most two active intervals, by splitting J at its midpoint j. We can think of this algorithm proceeding in levels, where at level λ we process those active intervals that have been obtained via λ splits. At each level, we sweep the intervals in a left-to-right manner, repeatedly merging consecutive intervals with the same label, so that the resulting interval inherits that label. Since in each level other than the first one each active interval contains an endpoint of the sought partition of R, we have at most $\mathcal{O}(\sqrt{n})$ active intervals in each level. As the sizes of active intervals decrease by a constant factor in each level, we have $\mathcal{O}(\log n)$ levels. We batch the $\mathcal{O}(\sqrt{n})$ queries for each level and answer them using Lemma 11. The total time required for partitioning R is thus $\mathcal{O}(\sqrt{n}\log^{1.5} n)$.

Next, we partition R by values $\mathsf{LPFpos}'_U[i]$ using the same algorithm in $\tilde{\mathcal{O}}(\sqrt{n})$ time. Finally, we compute an intersection of the two partitions, as desired, in $\mathcal{O}(\sqrt{n})$ time.

To update the data structure $\mathcal{D}$ using Lemma 20, we insert the edge from the old root to the new root using operation (a) and then perform the operation (b) on each of the $\mathcal{O}(\sqrt{n})$

groups of edges that are being redirected. In total, the data structure is updated in $\mathcal{O}(\sqrt{n}\pi_n)$ time, as desired. ◀

5.1 Implementation of a Semi-Dynamic LPF-tree using Link-cut Trees and Joinable Balanced BSTs

A link-cut tree is a classic data structure [61] that represents a forest of rooted trees containing n nodes in total. Each node stores an integer weight. The data structure has size $\mathcal{O}(n)$ and supports the following operations in amortized $\mathcal{O}(\log n)$ time:

- add a tree consisting of a single node to the forest;
- remove a tree consisting of a single node from the forest;
- attach a root node to another node as its child (*link* operation);
- given a node in one of the trees, disconnect it (and its subtree) from the tree of which it is part to form a separate tree (*cut* operation);
- add a given value α to the weights of all the descendants of a node;
- return the weight of a given node.

For implementations of operations including weights, see for example [40, Appendix: Splay trees and link-cut trees].

It is well-known (cf. [64, pp. 45-56]) that a collection of red-black trees (RB trees) containing n integer keys in total can support the following operations, each in $\mathcal{O}(\log n)$ time:

- insert an element to an RB tree;
- delete a given element from an RB tree;
- join two RB trees into one RB tree, provided that all keys in one of the trees are smaller than all keys in the other (the arguments of the join operation are not kept);
- split an RB tree into two RB trees, one containing all keys smaller than a specified parameter k and the other containing the remaining keys (again, the initial RB tree is not kept).

We note that every operation on an RB tree (adding a new leaf, removing a leaf, rotation) can be simulated using $\mathcal{O}(1)$ link and cut operations. We obtain the following observation.

▶ **Observation 23.** *A collection of RB trees on n nodes can be simulated using link-cut trees. The amortized cost of every operation on an RB tree is then $\mathcal{O}(\log^2 n)$.*

The semi-dynamic LPF-tree is represented using link-cut trees. A straightforward implementation would be sufficient to cover first-letter deletions (Lemma 21). However, when a new letter is appended to the string, the total number of single-edge updates could be $\Theta(n)$. To guarantee efficiency, edges need to be redirected in batches (Lemma 22). To this end, we use joinable balanced BSTs, such as RB trees, to represent all edges leading to a single node.

More formally, the whole LPF-tree is stored in one link-cut tree. There is a 1-to-1 correspondence between nodes of the LPF-tree and nodes of the link-cut tree. Assume node v of the LPF-tree is not a leaf and that the sources of all edges with target v are $u_1, u_2, \ldots, u_p$, with $u_1 < u_2 < \cdots < u_p$. Then the link-cut tree arranges the nodes $u_1, u_2, \ldots, u_p$ into an RB-tree and the root of this tree is joined with an edge with v.

In Lemma 21, we need to make $\mathcal{O}(\sqrt{n})$ single edge updates. Each of them requires the move of a node from one RB tree to another RB tree in our link-cut tree, which gives $\mathcal{O}(\sqrt{n}\log^2 n)$ time by Observation 23. In Lemma 22, in addition to operations on $\mathcal{O}(1)$ nodes and edges on the link-cut tree, we need to move a batch of consecutive nodes from an RB tree to a new RB tree leading to the new root. By Observation 23, the operations on the link-cut tree in this lemma require only $\mathcal{O}(\log^2 n)$ time. The resulting update time complexity $\mathcal{O}(\sqrt{n}\log^2 n)$ dominates the remaining operations from Lemmas 21 and 22.

We assume that edges inside RB trees have weight 0 and all the remaining edges have weight 1. Then, in the data structure the weights of nodes will be updated so that at each moment, the weight of a node will be equal to the sum of weights of edges on the path to the root. Then, for a node $i \in [a\,..\,a+|U|]$, where $a = \mathsf{deletions}(U)$, the weight will correspond to the length of the path in the LPF-tree from a to $a+|U|$. The weights can be maintained with $\mathcal{O}(1)$ "add" and "query" operations (which gives $\mathcal{O}(\log n)$ amortized time) per link or cut operation to satisfy this definition of weights. Indeed, no changes to weights are required in link or cut operations implementing a rotation on an RB tree; when moving a batch of edges to a different RB tree, we first query for the weights of the roots of the old RB tree and the new RB tree and then add the difference of these weights to the whole moved subtrees; adding a new root requires incrementing the weights of all the existing nodes.

By Lemmas 21 and 22, we obtain the following result, which together with Observation 18 implies our main result, Theorem 1.

▶ **Lemma 24.** *A semi-dynamic LPF-tree of a string of length at most n can be maintained in $\mathcal{O}(\sqrt{n}\log^2 n)$ amortized time per update operation, such that the length of the path from the root to any node can be retrieved in $\mathcal{O}(\log n)$ amortized time.*

6 Additive Sensitivity of LZ77 for Rotations

Note that as each rotation can be emulated with two edit operations, the work of Akagi et al. [2, Section 8] implies that, for any string S, $\frac{1}{6}|\mathsf{LZ}(\mathsf{rot}(S))| \le |\mathsf{LZ}(S)| \le 6|\mathsf{LZ}(\mathsf{rot}(S))|$.

▶ **Fact 25.** *There are infinitely many strings S for which $|LZ(\mathsf{rot}(S))| \ge |LZ(S)| + \Theta(\sqrt{|S|})$ and $|LZ(\mathsf{rot}(S))| \ge \frac{3}{2}|LZ(S)| - 2$.*

Proof. Let $a_1, \ldots, a_m$ be distinct letters and let $S_i = a_1 a_2 \cdots a_i$. Now, consider the string $S = S_m S_1 S_2 \cdots S_m$, which is of length $\Theta(m^2)$.

We have that $|\mathsf{LZ}(S)| = 2m$ since the phrases of $\mathsf{LZ}(S)$ are:

$$(a_1, a_2, \ldots, a_m, S_1, S_2, \ldots, S_m).$$

Let $S' = \mathsf{rot}(S) = a_2 a_3 \cdots a_m S_1 S_2 \cdots S_m a_1$. Then, we have $|\mathsf{LZ}(S')| = 3m - 2$ since the phrases of $\mathsf{LZ}(S')$ are:

$$(a_2, a_3, \ldots, a_m, a_1,\ S_1, a_2,\ S_2, a_3,\ S_3, a_4,\ \ldots, S_{m-2}, a_{m-1},\ S_{m-1}, a_m a_1).$$ ◀

The next fact provides a corresponding upper bound on $|\mathsf{LZ}(\mathsf{rot}(S))|$. Further, let us note that the rot operation can decrease the number of LZ phrases by one; for example, $|\mathsf{LZ}(abaa)| = 4$ and $|\mathsf{LZ}(baaa)| = 3$. Fact 26 also shows that a larger decrease is not possible.

▶ **Fact 26.** *For every string S, we have $|LZ(S)| - 1 \le |LZ(\mathsf{rot}(S))| \le |LZ(S)| + \Theta(\sqrt{|S|})$ and $|LZ(\mathsf{rot}(S))| \le 2|LZ(S)|$.*

Proof. The second inequality follows by Lemma 13. Indeed, let us consider the LZ parsing of S into phrases $F_1, F_2, \ldots, F_k$. We have $|F_1| = 1$. We can transform it into a parsing of $\mathsf{rot}(S)$ as follows: move F_1 to the end and for each phrase F_i whose previous occurrence was only at the leftmost position of S, partition F_i into a one-letter phrase and the remaining phrase. By Lemma 13, $\mathcal{O}(\sqrt{|S|})$ phrases will be partitioned. The resulting parsing cannot have fewer phrases than the LZ parsing of $\mathsf{rot}(S)$ by the fact that greedy is optimal in this case. The parsing has size $|\mathsf{LZ}(S)| + \Theta(\sqrt{|S|})$ and, simultaneously, size at most $2|\mathsf{LZ}(S)|$, as required.

Let us prove the first inequality. Let S' be S without its first letter. If suffices to show that $|\mathsf{LZ}(S')| \geq |\mathsf{LZ}(S)| - 1$, as appending letters can only increase the number of phrases.

For a position a in S, by $next_S(a)$ we denote the smallest ending position b of a phrase in $\mathsf{LZ}(S)$ such that $b \geq a$. We show by induction that if i is the ending position of a phrase in $\mathsf{LZ}(S')$, then $|\mathsf{LZ}(S'[0\,..\,i])| \geq |\mathsf{LZ}(S[0\,..\,j])| - 1$ for $j = next_S(i+1)$.

The base case for $i = 0$ holds with equality for $j = next_S(1)$. Assume that $|\mathsf{LZ}(S'[0\,..\,i])| \geq |\mathsf{LZ}(S[0\,..\,j])| - 1$ holds for i being an ending position of a phrase in S' and $j = next_S(i+1)$. Let $i' > i$ be the next ending position of a phrase in S' and $j' = next_S(i'+1)$. If $j' = j$, then the desired inequality holds as $|\mathsf{LZ}(S'[0\,..\,i'])| = |\mathsf{LZ}(S'[0\,..\,i])| + 1$ and $|\mathsf{LZ}(S[0\,..\,j'])| = \mathsf{LZ}(S[0\,..\,j])|$. Otherwise, we have that $j' = j''$, where $j'' = next_S(j+1)$: Indeed, if $j' > j''$, then $S'[j\,..\,j'-1]$ would have an earlier occurrence in S', so $S[j+1\,..\,j'] = S'[j\,..\,j'-1]$ would have an earlier occurrence in S, which contradicts the greediness of the algorithm computing the parsing. By the inductive assumption, this concludes that

$$|\mathsf{LZ}(S'[0\,..\,i'])| = |\mathsf{LZ}(S'[0\,..\,i])|+1 \geq |\mathsf{LZ}(S[0\,..\,j])| = |\mathsf{LZ}(S[0\,..\,j''])|-1 = |\mathsf{LZ}(S[0\,..\,j'])|-1,$$

as required. ◀

7 Conclusions

We have shown that the size of the Lempel-Ziv-Storer-Szymanski factorization (LZSS) with self-references of a length-n semi-dynamic string S can be updated in $\tilde{\mathcal{O}}(\sqrt{n})$ time. The same approach with minor adaptations can store the size of the classic LZ77 parsing (with self-references), in which the phrase of S starting at position i is $S[i\,..\,\min(i+\mathsf{LPF}_S[i], |S|-1)]$ (i.e., it includes the position that immediately follows the longest previous factor), also with $\tilde{\mathcal{O}}(\sqrt{n})$ update time. Future work includes storing the size of other types of compression in the semi-dynamic setting. The main open problems are, however, if the update time can be decreased and if strictly sublinear update time is possible in the fully dynamic setting.

References

1 Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004. doi:10.1016/S1570-8667(03)00065-0.

2 Tooru Akagi, Mitsuru Funakoshi, and Shunsuke Inenaga. Sensitivity of string compressors and repetitiveness measures. *Information and Computation*, 291:104999, 2023. doi:10.1016/J.IC.2022.104999.

3 Tooru Akagi, Yuki Kuhara, Takuya Mieno, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Combinatorics of minimal absent words for a sliding window. *Theoretical Computer Science*, 927:109–119, 2022. doi:10.1016/J.TCS.2022.06.002.

4 Anisa Al-Hafeedh, Maxime Crochemore, Lucian Ilie, Evguenia Kopylova, William F. Smyth, German Tischler, and Munina Yusufu. A comparison of index-based Lempel-Ziv LZ77 factorization algorithms. *ACM Computing Surveys*, 45(1):5:1–5:17, 2012. doi:10.1145/2379776.2379781.

5 Arne Andersson and Mikkel Thorup. Dynamic ordered sets with exponential search trees. *Journal of the ACM*, 54(3):13, 2007. doi:10.1145/1236457.1236460.

6 Hideo Bannai, Travis Gagie, and Tomohiro I. Refining the r-index. *Theoretical Computer Science*, 812:96–108, 2020. doi:10.1016/J.TCS.2019.08.005.

7 Djamal Belazzougui, Dmitry Kosolobov, Simon J. Puglisi, and Rajeev Raman. Weighted ancestors in suffix trees revisited. In *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021*, pages 8:1–8:15, 2021. doi:10.4230/LIPICS.CPM.2021.8.

8 Djamal Belazzougui and Simon J. Puglisi. Range predecessor and Lempel-Ziv parsing. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016*, pages 2053–2071, 2016. doi:10.1137/1.9781611974331.CH143.

9 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium*, pages 88–94. Springer, 2000. doi:10.1007/10719839_9.

10 Philip Bille, Patrick Hagge Cording, Johannes Fischer, and Inge Li Gørtz. Lempel-Ziv compression in a sliding window. In *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017*, pages 15:1–15:11, 2017. doi:10.4230/LIPICS.CPM.2017.15.

11 Dany Breslauer and Zvi Galil. Finding all periods and initial palindromes of a string in parallel. *Algorithmica*, 14(4):355–366, 1995. doi:10.1007/BF01294132.

12 Andrej Brodnik and Matevz Jekovec. Sliding suffix tree. *Algorithms*, 11(8):118, 2018. doi:10.3390/A11080118.

13 Graham Cormode and S. Muthukrishnan. Substring compression problems. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005*, pages 321–330. SIAM, 2005. URL: http://dl.acm.org/citation.cfm?id=1070432.1070478.

14 Maxime Crochemore. Linear searching for a square in a word. *Bulletin-European Association for Theoretical Computer Science*, 24(1):66–72, 1984.

15 Maxime Crochemore, Alice Héliou, Gregory Kucherov, Laurent Mouchard, Solon P. Pissis, and Yann Ramusat. Absent words in a sliding window with applications. *Information and Computation*, 270, 2020. doi:10.1016/J.IC.2019.104461.

16 Maxime Crochemore and Lucian Ilie. Computing longest previous factor in linear time and applications. *Information Processing Letters*, 106(2):75–80, 2008. doi:10.1016/J.IPL.2007.10.006.

17 Maxime Crochemore, Lucian Ilie, Costas S. Iliopoulos, Marcin Kubica, Wojciech Rytter, and Tomasz Waleń. Computing the longest previous factor. *European Journal of Combinatorics*, 34(1):15–26, 2013. doi:10.1016/J.EJC.2012.07.011.

18 Maxime Crochemore, Lucian Ilie, and William F. Smyth. A simple algorithm for computing the Lempel Ziv factorization. In *2008 Data Compression Conference (DCC 2008)*, pages 482–488, 2008. doi:10.1109/DCC.2008.36.

19 Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, M. Sohel Rahman, German Tischler, and Tomasz Waleń. Improved algorithms for the range next value problem and applications. *Theoretical Computer Science*, 434:23–34, 2012. doi:10.1016/J.TCS.2012.02.015.

20 Jonas Ellert. Sublinear time Lempel-Ziv (LZ77) factorization. In *String Processing and Information Retrieval - 30th International Symposium, SPIRE 2023*, pages 171–187, 2023. doi:10.1007/978-3-031-43980-3_14.

21 Edward R. Fiala and Daniel H. Greene. Data compression with finite windows. *Communications of the ACM*, 32(4):490–505, 1989. doi:10.1145/63334.63341.

22 Nathan J. Fine and Herbert S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965. doi:10.2307/2034009.

23 Johannes Fischer and Pawel Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In *Combinatorial Pattern Matching, CPM 2015*, pages 160–171, 2015. doi:10.1007/978-3-319-19929-0_14.

24 Johannes Fischer, Tomohiro I, and Dominik Köppl. Lempel Ziv computation in small space (LZ-CISS). In *Combinatorial Pattern Matching - 26th Annual Symposium, CPM 2015*, pages 172–184, 2015. doi:10.1007/978-3-319-19929-0_15.

25 Johannes Fischer, Tomohiro I, Dominik Köppl, and Kunihiko Sadakane. Lempel-Ziv factorization powered by space efficient suffix trees. *Algorithmica*, 80(7):2048–2081, 2018. doi:10.1007/S00453-017-0333-1.

26 Younan Gao, Meng He, and Yakov Nekrich. Fast preprocessing for optimal orthogonal range reporting and range successor with applications to text indexing. In *28th Annual European*

Symposium on Algorithms, ESA 2020, pages 54:1–54:18, 2020. doi:10.4230/LIPICS.ESA.2020.54.

27 Keisuke Goto and Hideo Bannai. Simpler and faster Lempel Ziv factorization. In *2013 Data Compression Conference, DCC 2013*, pages 133–142, 2013. doi:10.1109/DCC.2013.21.

28 Keisuke Goto and Hideo Bannai. Space efficient linear time Lempel-Ziv factorization for small alphabets. In *Data Compression Conference, DCC 2014*, pages 163–172, 2014. doi:10.1109/DCC.2014.62.

29 Shunsuke Inenaga, Ayumi Shinohara, Masayuki Takeda, and Setsuo Arikawa. Compact directed acyclic word graphs for a sliding window. *Journal of Discrete Algorithms*, 2(1):33–51, 2004. doi:10.1016/S1570-8667(03)00064-9.

30 Yusuke Ishida, Shunsuke Inenaga, Ayumi Shinohara, and Masayuki Takeda. Fully incremental LCS computation. In *15th International Symposium on Fundamentals of Computation Theory, FCT 2005*, pages 563–574, 2005. doi:10.1007/11537311_49.

31 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lightweight Lempel-Ziv parsing. In *Experimental Algorithms, 12th International Symposium, SEA 2013*, volume 7933, pages 139–150, 2013. doi:10.1007/978-3-642-38527-8_14.

32 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In *Combinatorial Pattern Matching, 24th Annual Symposium, CPM 2013*, pages 189–200, 2013. doi:10.1007/978-3-642-38905-4_19.

33 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lempel-Ziv parsing in external memory. In *Data Compression Conference, DCC 2014*, pages 153–162, 2014. doi:10.1109/DCC.2014.78.

34 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006. doi:10.1145/1217856.1217858.

35 Orgad Keller, Tsvi Kopelowitz, Shir Landau Feibish, and Moshe Lewenstein. Generalized substring compression. *Theoretical Computer Science*, 525:42–54, 2014. doi:10.1016/J.TCS.2013.10.010.

36 Dominik Kempa. Optimal construction of compressed indexes for highly repetitive texts. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019*, pages 1344–1357, 2019. doi:10.1137/1.9781611975482.82.

37 Dominik Kempa and Tomasz Kociumaka. Dynamic suffix array with polylogarithmic queries and updates. In *STOC 2022: 54th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1657–1670, 2022. doi:10.1145/3519935.3520061.

38 Dominik Kempa and Simon J. Puglisi. Lempel-Ziv factorization: Simple, fast, practical. In *Proceedings of the 15th Meeting on Algorithm Engineering and Experiments, ALENEX 2013*, pages 103–112, 2013. doi:10.1137/1.9781611972931.9.

39 Sung-Ryul Kim and Kunsoo Park. A dynamic edit distance table. *Journal of Discrete Algorithms*, 2(2):303–312, 2004. doi:10.1016/S1570-8667(03)00082-0.

40 Philip Klein and Shay Mozes. *Optimization Algorithms for Planar Graphs*. 2023. URL: https://planarity.org/.

41 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977. doi:10.1137/0206024.

42 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal pattern matching queries in a text and applications. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 532–551, 2015. doi:10.1137/1.9781611973730.36.

43 Dominik Köppl. Non-overlapping LZ77 factorization and LZ78 substring compression queries with suffix trees. *Algorithms*, 14(2):44, 2021. doi:10.3390/A14020044.

44 Dominik Köppl and Kunihiko Sadakane. Lempel-Ziv computation in compressed space (LZ-CICS). In *2016 Data Compression Conference, DCC 2016*, pages 3–12, 2016. doi:10.1109/DCC.2016.38.

45 Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt. Incremental string comparison. *SIAM Journal on Computing*, 27(2):557–582, 1998. doi:10.1137/S0097539794264810.

46 N. Jesper Larsson. Extended application of suffix trees to data compression. In *Proceedings of the 6th Data Compression Conference (DCC 1996)*, pages 190–199, 1996. doi:10.1109/DCC.1996.488324.

47 Takuya Mieno, Yuta Fujishige, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing minimal unique substrings for a sliding window. *Algorithmica*, 84(3):670–693, 2022. doi:10.1007/S00453-021-00864-1.

48 Takuya Mieno and Mitsuru Funakoshi. Shortest unique palindromic substring queries in semi-dynamic settings. In *Combinatorial Algorithms - 33rd International Workshop, IWOCA 2022*, pages 425–438, 2022. doi:10.1007/978-3-031-06678-8_31.

49 Takuya Mieno, Kiichi Watanabe, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Palindromic trees for a sliding window and its applications. *Information Processing Letters*, 173:106174, 2022. doi:10.1016/J.IPL.2021.106174.

50 Yakov Nekrich and Gonzalo Navarro. Sorted range reporting. In *Algorithm Theory - SWAT 2012 - 13th Scandinavian Symposium and Workshops*, pages 271–282, 2012. doi:10.1007/978-3-642-31155-0_24.

51 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Fully dynamic data structure for LCE queries in compressed space. In *41st International Symposium on Mathematical Foundations of Computer Science, MFCS 2016*, pages 72:1–72:15, 2016. doi:10.4230/LIPICS.MFCS.2016.72.

52 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Fully dynamic data structure for LCE queries in compressed space. *CoRR*, abs/1605.01488, 2016. arXiv:1605.01488.

53 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Dynamic index and LZ factorization in compressed space. *Discrete and Applied Mathematics*, 274:116–129, 2020. doi:10.1016/J.DAM.2019.01.014.

54 Enno Ohlebusch and Simon Gog. Lempel-Ziv factorization revisited. In *Combinatorial Pattern Matching - 22nd Annual Symposium, CPM 2011*, pages 15–26, 2011. doi:10.1007/978-3-642-21458-5_4.

55 Daisuke Okanohara and Kunihiko Sadakane. An online algorithm for finding the longest previous factors. In *Algorithms - ESA 2008, 16th Annual European Symposium*, pages 696–707, 2008. doi:10.1007/978-3-540-87744-8_58.

56 Wojciech Plandowski and Wojciech Rytter. Application of Lempel-Ziv encodings to the solution of words equations. In *Automata, Languages and Programming, 25th International Colloquium, ICALP 1998*, pages 731–742, 1998. doi:10.1007/BFB0055097.

57 Alberto Policriti and Nicola Prezza. LZ77 computation based on the run-length encoded BWT. *Algorithmica*, 80(7):1986–2011, 2018. doi:10.1007/S00453-017-0327-Z.

58 Nicola Prezza and Giovanna Rosone. Faster online computation of the succinct longest previous factor array. In *Beyond the Horizon of Computability - 16th Conference on Computability in Europe, CiE 2020*, pages 339–352, 2020. doi:10.1007/978-3-030-51466-2_31.

59 Milan Ružić. Constructing efficient dictionaries in close to sorting time. In *International Colloquium on Automata, Languages and Programming, ICALP 2008*, pages 84–95. Springer, 2008. doi:10.1007/978-3-540-70575-8_8.

60 Martin Senft and Tomás Dvorák. Sliding CDAWG perfection. In *String Processing and Information Retrieval, 15th International Symposium, SPIRE 2008*, pages 109–120, 2008. doi:10.1007/978-3-540-89097-3_12.

61 Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983. doi:10.1016/0022-0000(83)90006-5.

62 Tatiana Starikovskaya. Computing Lempel-Ziv factorization online. In *Mathematical Foundations of Computer Science 2012 - 37th International Symposium, MFCS 2012*, pages 789–799, 2012. doi:10.1007/978-3-642-32589-2_68.

63 James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, 1982. doi:10.1145/322344.322346.

64 Robert Endre Tarjan. *Data structures and network algorithms*, volume 44 of *CBMS-NSF regional conference series in applied mathematics*. SIAM, 1983. doi:10.1137/1.9781611970265.

65 Mikkel Thorup. Equivalence between priority queues and sorting. *Journal of the ACM*, 54(6):28, 2007. doi:10.1145/1314690.1314692.

66 Alexandre Tiskin. Semi-local string comparison: algorithmic techniques and applications. *CoRR*, abs/0707.3619, 2007. arXiv:0707.3619.

67 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.

68 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983. doi:10.1016/0020-0190(83)90075-3.

69 Jun-ichi Yamamoto, Tomohiro I, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Faster compact on-line Lempel-Ziv factorization. In *31st International Symposium on Theoretical Aspects of Computer Science, STACS 2014*, pages 675–686, 2014. doi:10.4230/LIPICS.STACS.2014.675.

70 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977. doi:10.1109/TIT.1977.1055714.